

Transaction-based simulation using SystemC/SCV

By Eisuke Yuri

Engineer
Fujitsu Network Technologies Ltd
E-mail: yuri.eisuke@jp.fujitsu.com

Junichi Tatsuda

Lead Sales Application Engineer
Incisive Core Competency Group
Cadence Design Systems
E-mail: tatsuda@cadence.com

Neyaz Khan

Verification Architect
Incisive Core Competency Group
Cadence Design Systems
E-mail: nkhan@cadence.com

Chris Dietrich

Director
Incisive Core Competency Group
Cadence Design Systems
E-mail: chris@cadence.com

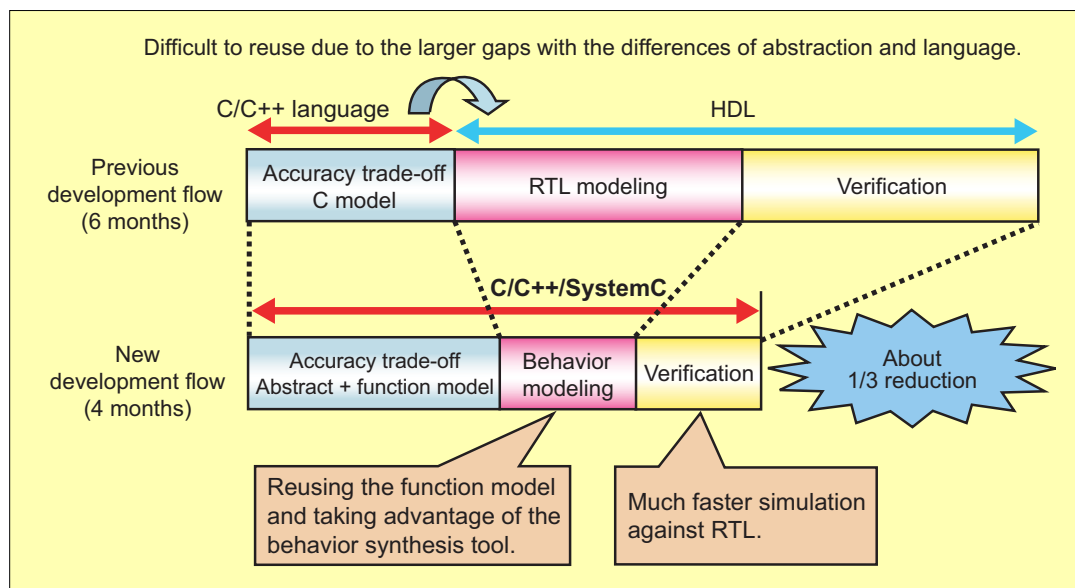


Figure 1: SystemC is used to reduce development effort. The diagram compares two high-level project development flows: architectural exploration and a detailed cycle-accurate RTL design and functional verification.

Fujitsu Network Technologies Ltd has accomplished two successful projects using the SystemC verification library (SCV) with Cadence Design System's incisive unified simulator (IUS). Fujitsu's team of engineers chose the IUS in creating an integrated environment because it has the ability to do multilingual simulation using SystemC and mixed hardware description languages (HDLs). The single environment allowed both design and verification engineers to seamlessly share the test and module IP, thus eliminating any rework.

Figure 1 compares two high-level project development flows: architectural exploration and a detailed cycle-accurate register-transfer level (RTL) design and functional verification.

The first one is a typical flow traditionally preferred by most companies. On the other hand, the initial development and verification flow entails developing a C-based algorithmic model to investigate feasibility at higher levels of abstraction. Once the architect is satisfied with the fixed-point C model, it is handed over to the HDL developer along with the spec as reference. Typically, C-

models are functionally accurate but have no timing information. They are also not re-entrant, which means that they do not store any previously computed data (e.g. value of an accumulator in an FIR filter). The design is then redeveloped at the RTL using an HDL. Subsequently, complete functional verification is also performed at the RTL.

Although the flow initially worked, the team soon realized its problems. For one, the C

module was not specifically being reused in either the design or test of the RTL implementation. Moreover, the design was being developed and verified twice, once at the C level and once at the relatively slower RTL. Bugs were introduced into the RTL implementation because of misinterpretations by the RTL developer, typically when introducing design time and cycle accuracy at the detailed signal-level implementation. With the advent of

SystemC and its associated utilities and methodologies, much of these inefficiencies could be eliminated.

The new flow heavily utilized the notion of reuse to cut overall design and verification time by spending more time up front, developing and verifying a model that can be reused throughout the RTL implementation phase. This provided the RTL developers with an executable spec and reusable testbenches, thus elimi-

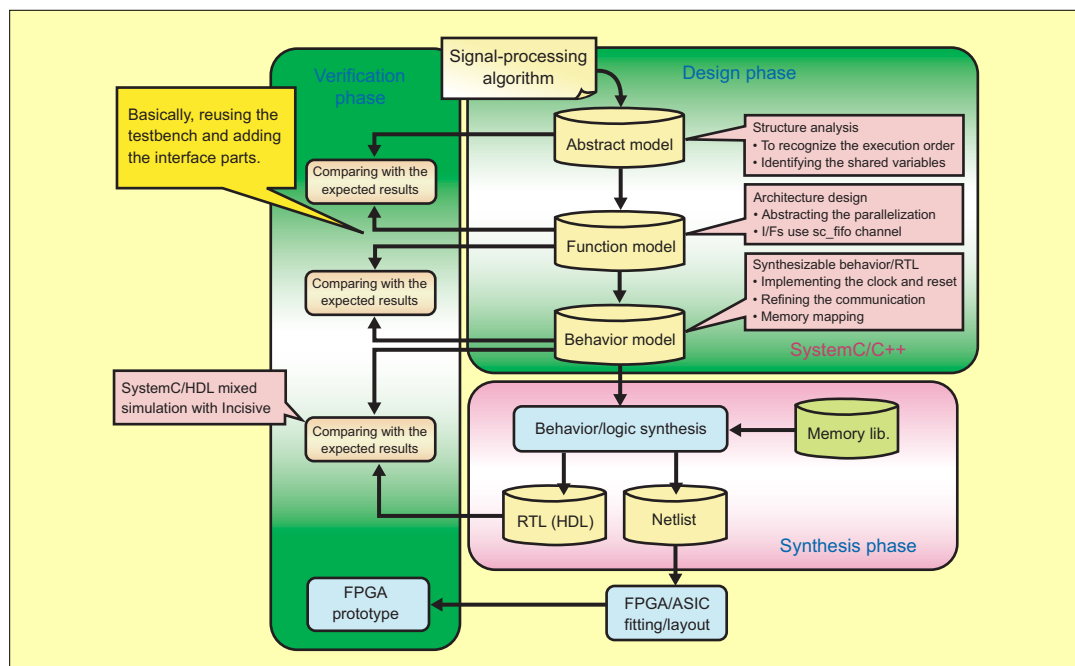


Figure 2: In the SystemC-based flow, module reuse is evident in both the verification and design phases.

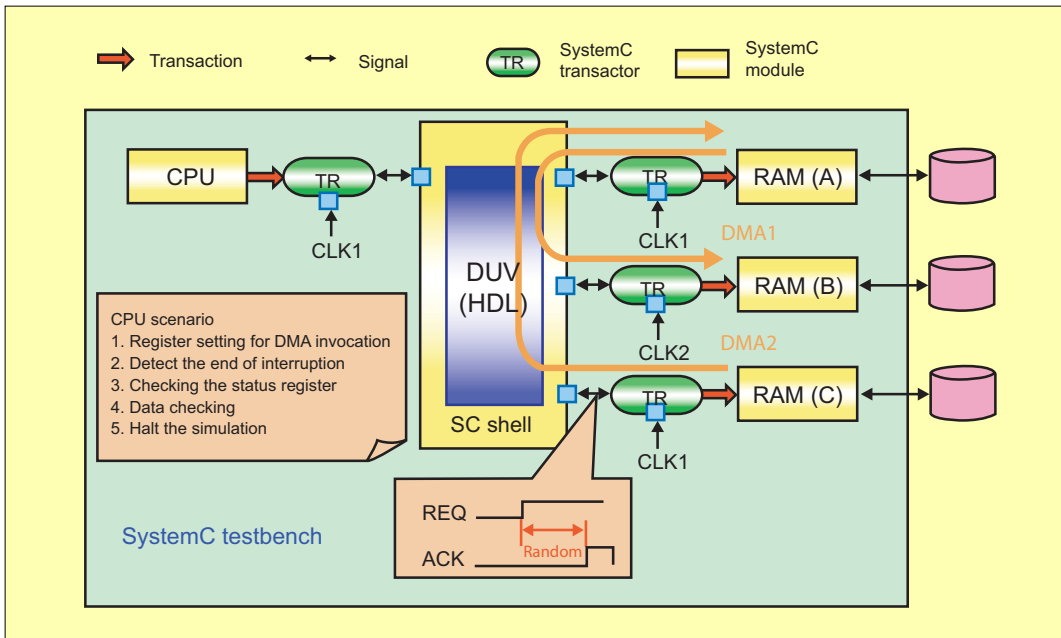


Figure 3: This graphical view of the test plan reveals the test goals and analysis output.

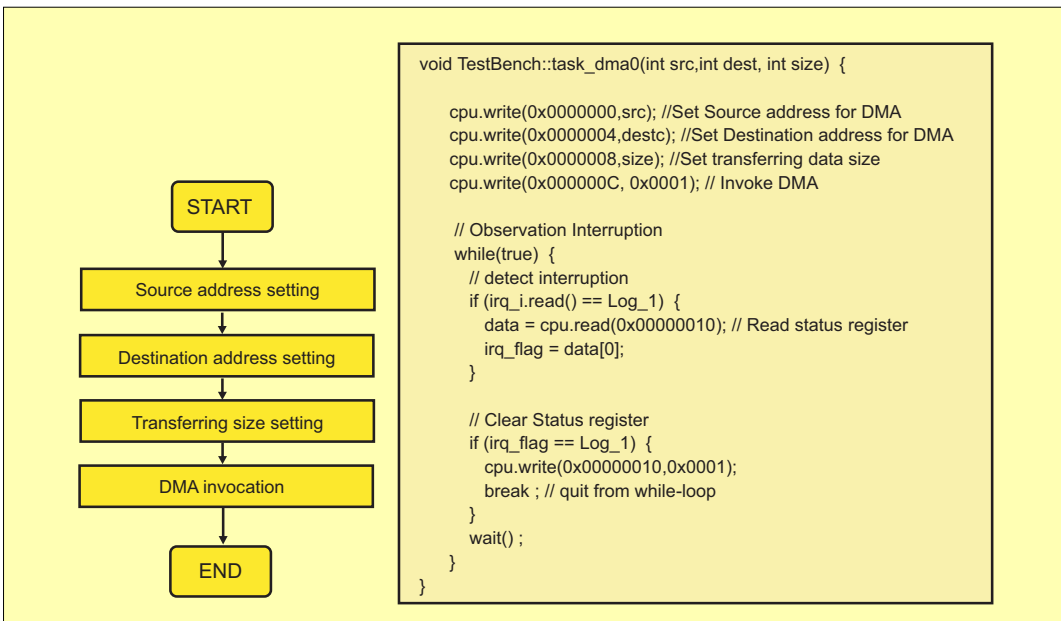


Figure 4: Verilog DMA invocation sequence (pseudo-code) describes a function in the testbench initializing the CPU module to drive data to and from the DUV (DMA), forcing an interrupt in the DMA.

nating possibilities of any mis-interpretation. Less time in RTL design and verification was needed, thus reducing overall project time by 30 percent.

Figure 2 details how the new flow was implemented. Module reuse was evident in both the verification and design phases. Within the design phase, models were developed, verified, tested and reused at various levels of abstraction.

The signal-processing algorithm started at a very high level of abstraction. Concepts were first tried out using floating point C/C++, and then fixed-point C was used and the algorithm verified for targeted functionality and performance. Once the architects

had achieved their desired performance, the algorithm was mapped to an appropriate microarchitecture that was implemented as RTL by the design team. This stage of the process traditionally had many problems due to the gaps in abstraction between C/C++ and HDL. It was a rich source of functional bugs in the design. Thus, it was highly suggested to spend extra time at this stage to introduce timing and bit-accurate performance against highly abstract system-level models.

Appropriate functionality at each level of abstraction was verified efficiently to enable architectural/structural decisions. As the models were now

reused throughout the design process and into the RTL development phase, these critical decisions no longer had to be redeveloped by the RTL de-

signer. By using the SystemC-based environment, RTL can be directly verified at each level of abstraction against reference models with the same intent and approximations as those used by system architects in the algorithmic and microarchitecture development phase.

For functional verification, the testbench was reused throughout the entire process with only transactor-interface modules needing enhancement to accommodate the increase in modeling complexity. Again, reuse of the testbench modules between SystemC and RTL greatly reduced the need for redevelopment and ensured design intent was maintained throughout the project cycle, while minimizing redundancy in the overall development effort.

The two case studies are isolated examples developed by the team for this design using advanced SystemC/SCV features:

Transactors as timing interface

The first project involved the design under verification written in Verilog and other parts of the testbench in SystemC/SCV. Transaction-level models (TLMs) of the CPU and RAM were written in SystemC. This section describes how transactors served as timing interface between RTL and TLMs and how CPU test scenarios were generated and simulated—including testbench random constraints for concurrent and random DMA sequences, RAM acknowledge returned with random delay across multiple clock domains, and data load and dump with randomly programmed delays.

A graphical view of the test

```

const unsigned SRC_MIN      = 0x55555555; // Source address [MIN]
const unsigned SRC_MAX      = 0xAAAAAAAA; // Source address [MAX]
const unsigned DST_MIN      = 0x33333333; // Destination address [MIN]
const unsigned DST_MAX      = 0xCCCCCCCC; // Destination address [MAX]
const unsigned SIZE_MIN     = 8; // Transferring size [MIN]
const unsigned SIZE_MAX     = 64; // Transferring size [MAX]
////////////////////////////////////

// Declare smart pointer
scv_smart_ptr<unsigned> intervalP; // Interval time between DMA invocations
scv_smart_ptr<unsigned> srcP; // Source address
scv_smart_ptr<unsigned> dstP; // destination address
scv_smart_ptr<unsigned> sizeP; // Transferring size

// Constraint
srcP->keep_only(SRC_MIN,SRC_MAX);
dstP->keep_only(DST_MIN,SRC_MAX);
sizeP->keep_only(SIZE_MIN,SIZE_MAX);
intervalP->keep_only(0,10);

```

Figure 5: SystemC verification constraints are set for a desired traffic pattern to be generated.

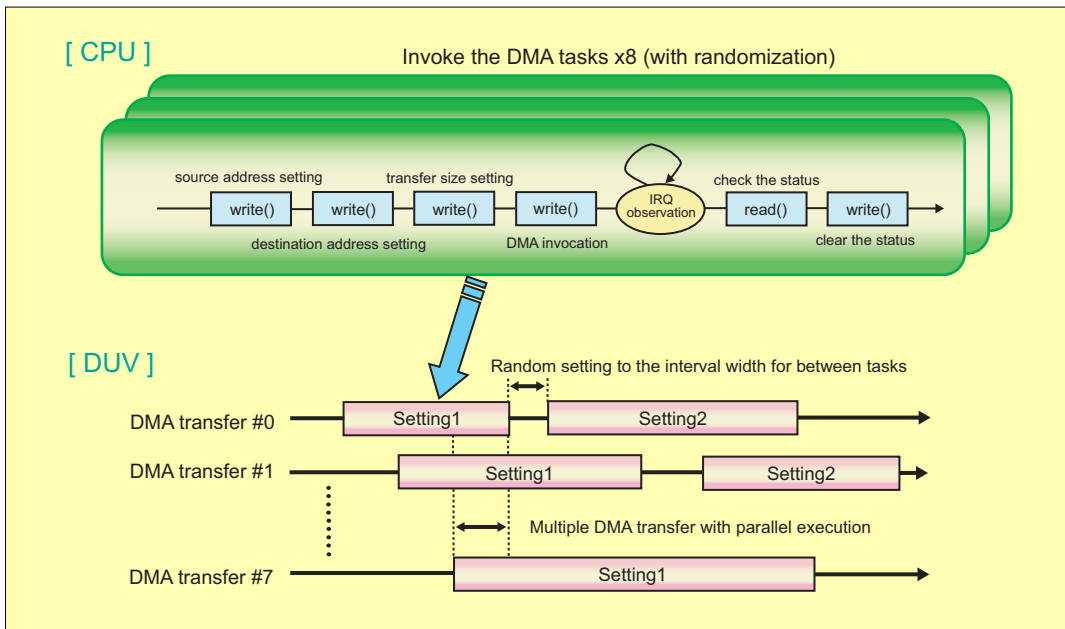


Figure 6: DMA write/reads happen with the random data and intervals and are also being issued in parallel threads, exercising the DMA's ability to handle multiple concurrent requests.

plan (Figure 3) reveals the following test goals and analysis output:

1. Randomly-generated CPU commands to device under test for DMA transfers, random number of DMA transfers based on specified constraints—Maximum of eight transfers.
2. Acknowledge from RAM generated with random delay based on programmable constraints—Acknowledge with random time (0-30clk).
3. Dump the RAM data based on CPU commands.
4. Create transactions for analysis—All DMA traffic (Figure 8).

The test plan calls for a random set of CPU commands—writing and reading data to and from the DMA. SCV randomization is used to invoke the CPU/DMA transactions. Randomization provides a more realistic set of test data scenarios exercising the DMA and its registers.

The task of actually driving the DMA is developed as a callable function. The pseudo-code in Figure 4 is describing a function in the testbench initializing the CPU module to drive data to and from the DUV (DMA), forcing an interrupt in the DMA.

For the set of randomized data to be meaningful, they need to be constrained to remain within the test plan objectives and design parameters. Figure 5 shows how con-

straints were set for the desired traffic pattern to be generated—source/destination addresses, packet sizes and random REQ/ACK intervals were all constrained to remain within the specified ranges using smart-pointers and SCV constraints.

Figure 6 shows the outcome of the above. In reviewing the anticipated execution sequence, we can see that not only were the DMA write/reads happening with the random data and intervals, they were also being issued in parallel threads, exercising the DMA's ability to handle multiple concurrent requests.

The team subsequently implemented transaction recordings such that the anticipated sequences could be captured and easily visualized in

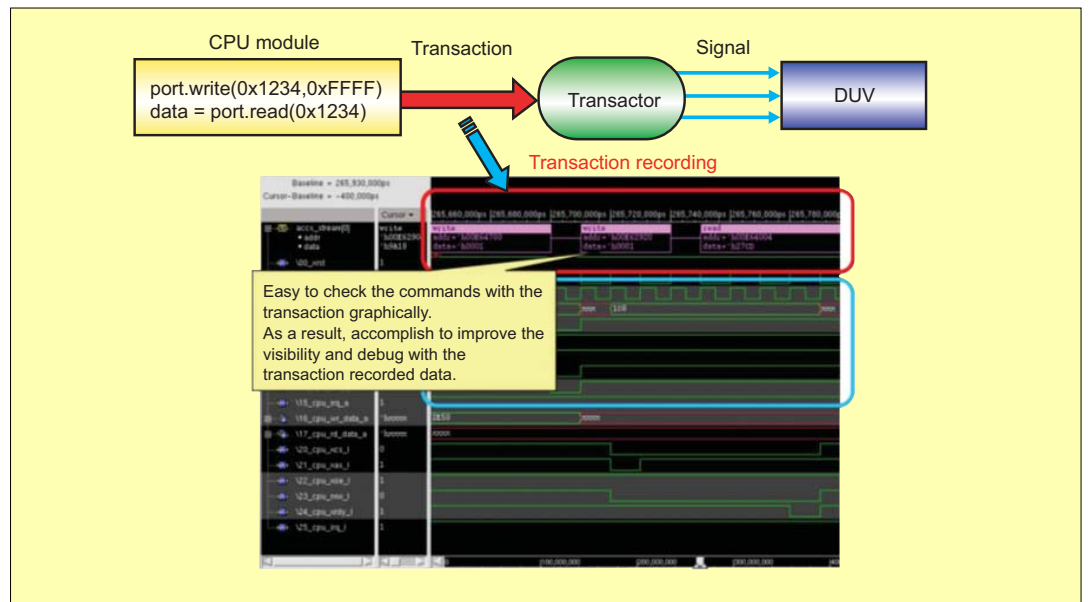


Figure 7: Using SystemC/SCV transaction-recording features and embedded SDI calls within the SCV, the resulting simulations indeed show that the anticipated test sequences were executing correctly.

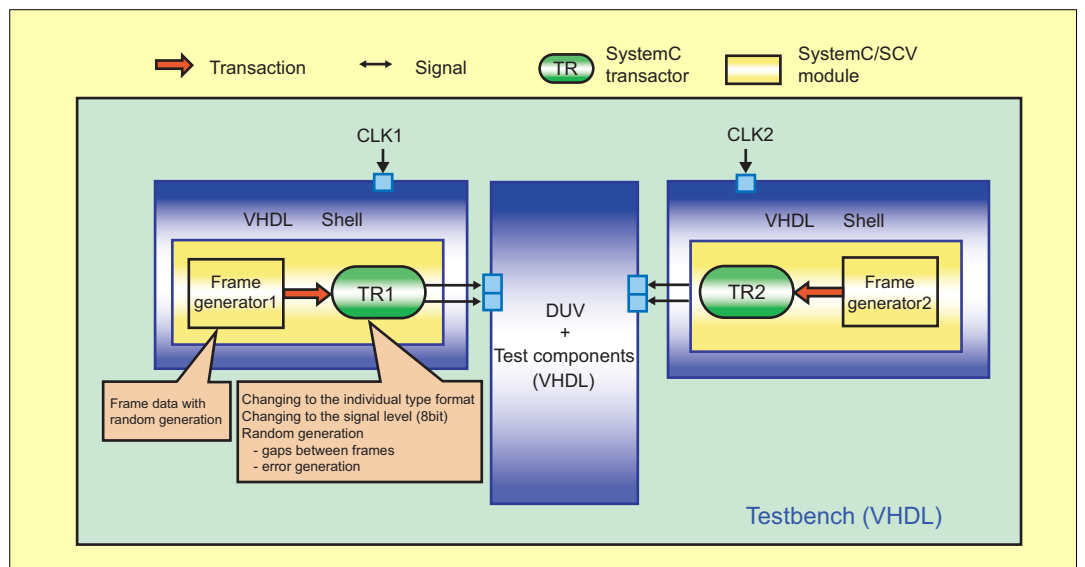


Figure 8: Frame data is submitted upstream and compared downstream. SystemC verification modules are wrapped in VHDL to provide seamless signal level integration with the DUV.

```

--Here transaction gen_frame_i is converted transaction level to signals.
SC_MODULE(IP_frame_conv) {
// Transaction Interface
sc_fifo_in<IP_general_t> gen_frame_i; // Transaction of IP_frame
// Signal Interface
sc_out<GMII_data_t> data_o;
sc_out<bool> enb_o; // data enable
sc_out<bool> err_o; // data error
sc_in<clk> clk; // clock

// member data
scv_smart_ptr<bool> trans_err; // ***** smart pointer for data error
scv_bag<bool> error_rate; // ***** error rate

// Constructor
SC_CTOR(IP_frame_conv) {

// randomization constraint
// To generate the data error at 1/1000 rate here
error_rate.add(true, 1);
error_rate.add(false, 1000-1);
trans_err->set_mode(error_rate);
}
}

```

Figure 9: For each frame generated, SCV randomized four elements of a frame packet.

the waveform viewer when simulating in the IUS. Using SystemC/SCV transaction-recording features and embedded SDI (transaction recording) calls within the SCV, the resulting simulations showed that the anticipated test sequences were executing correctly (Figure 7).

Transactions in the IUS platform are a simple and efficient way to record cause-and-effect sequences during a simulation run. When a problem in simulation is detected by the verification engineer, it is just a matter of reviewing the associated transactions to trace back to the origin of the problem. Any existing relationship between transactions—e.g. child-transactions spawned from a parent-transaction or error-transactions—can be analyzed using the explorer tool.

Dual frame generator TLMs
The second case study describes a test environment consisting of two frame generator TLMs that send random traffic

to a VHDL design under verification. Engineers felt the necessity to somewhat stress-test the randomization capabilities of SCV from within VHDL before committing to full-scale test development. Similar to the first project, the randomization capabilities of SCV were used to create hierarchical constraints and generate random test parameters with the following objectives: random frame length and data; random frame gaps; and random error packets based on defined distribution.

Frame data was submitted upstream and compared downstream (Figure 8). Note that the SystemC verification modules were wrapped in VHDL to provide seamless signal-level integration with the DUV.

For each frame generated,

SCV randomizes four elements of a frame packet, creating a realistic scenario. Randomizing the header, data, source/test addresses and frame intervals enabled the frames to produce not only realistic traffic, but also error packets within a desired probability of 1 in 1,000 frames (Figure 9).

The team has concluded that SystemC can provide them with performance and flexibility to model algorithms to hardware, all within the same environment. Since the language is suitable for multiple design and verification tasks, they were able to implement their code for reuse throughout the design cycle. For debugging designs, transaction-level verification provided higher levels of performance, visibility and cycle. □

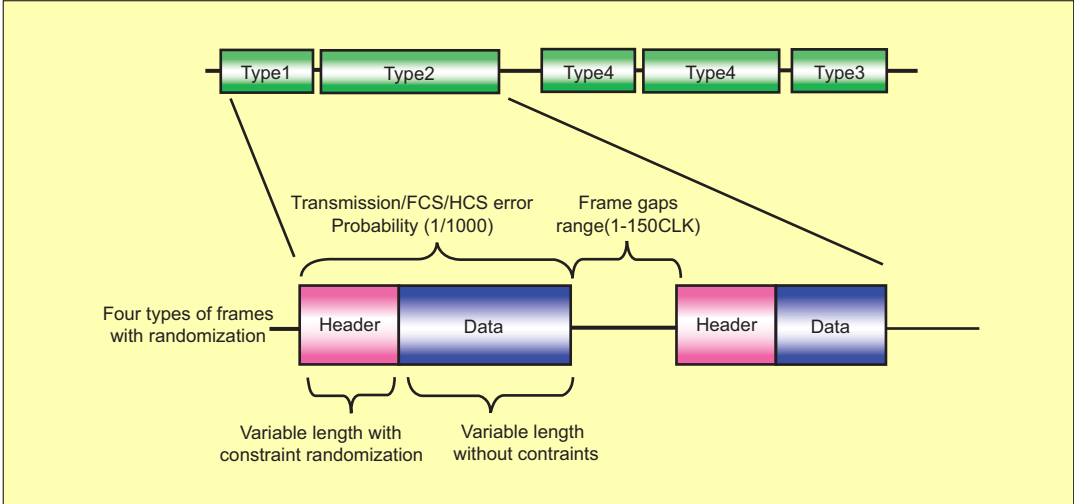


Figure 10: Randomizing the header, data, source/des addresses and frame intervals enabled the frames to produce not only realistic traffic, but error packets within a desired probability of 1 in 1,000 frames.